Feature Articles

Dynamic Programming

Thursday, April 1, 2004

By vorthys
TopCoder Member

Introduction

If you were to survey the reds, you would probably find that 9 out of 10 of them recommend the study of dynamic programming as one of the best ways to improve your rating... and the 10th doesn't only because he makes it a policy never to help future competitors! Dynamic programming (hereafter known as DP) is an algorithmic technique applicable to many counting and optimization problems. So, if you see the words "how many" or "minimum" or "maximum" or "shortest" or "longest" in a problem statement, chances are good that you're looking at a DP problem!

It All Starts with Recursion

I'll begin with the bad news. To understand DP, you must first have a firm grasp on recursion. If the very thought of recursion makes that rash on your neck come back, then stop reading now and go write a few for-loops until it goes away. The good news is that, once you've thoroughly mastered DP, a solution will often be just that—a few for-loops. But there's simply no way to reach that blissful state of enlightenment without passing through recursion along the way.

Actually, the recursive relationships found in dynamic programming are often really easy, because they are allowed to be inefficient. Spectacularly inefficient. Obscenely inefficient. The efficiency will come later.

As an example of how straightforward these recursive relationships can be, consider the **longest common subsequence** problem. You are given two strings, S and T, and want to find the longest subsequence that appears in both strings, where the characters of the subsequence do not have to appear consecutively in the original strings. For example, given the strings "ABCDE" and "DACACBE", the longest common subsequence is "ACE". The straightforward recursive algorithm can be written in pseudocode as

```
function LCS(S, T) is
if S is empty or T is empty then return empty string
if first char of S == first char of T then
return (first char of S) + LCS(S - first char, T - first char)
otherwise // first chars are different
return longer of LCS(S - first char, T) and LCS(S, T - first char)
```

In the first case, one or both of the strings is empty, so the longest common subsequence is empty. In the second case, both strings begin with the same character, so we declare that character to be part of the longest common subsequence and recursively calculate the longest common subsequence of the remaining characters. In the third case, the strings begin with different characters, so at least one of the first character of S or the first character of T is **not** part of the longest common subsequence. Therefore, we recursively calculate the longest common subsequences that we would get by dropping the first character of S and by dropping the first character of T, and keep whichever answer is longer.

If—and I admit it's a big if—you are comfortable with recursion, then this definition should make perfect sense to you. However, the part of your brain that is attuned to 8-second time limits may be screaming "But that's exponential! It'll never run fast enough!" Well, hand your temporal lobe a cookie, because it's right. This algorithm is horribly slow! But that's okay, because we're not done yet. Next, we'll take a look at why this algorithm is so slow, which will give us a big clue about how to make it faster.

Overlapping Subproblems

Consider what happens in the recursive LCS algorithm on inputs such as "ABCDE" and "FGHI". The progress of the algorithm can be summarized as

```
LCS("ABCDE", "FGHI")
= longer of { LCS("BCDE","FGHI"),
LCS("ABCDE","GHI") }
= longer of {
longer of { LCS("CDE","FGHI"),
LCS("BCDE","GHI") },
longer of { LCS("BCDE","GHI"),
LCS("ABCDE","HI") }}
= longer of {
longer of {
longer of { LCS("DE","FGHI"), LCS("CDE","GHI") },
longer of { LCS("CDE","GHI"), LCS("BCDE","HI") }},
longer of {
longer of { LCS("CDE","GHI"), LCS("BCDE","HI") },
longer of { LCS("BCDE","HI"), LCS("ABCDE","I") }}}
= ...
```

Notice how certain subproblems appear over and over again. Already we see three calls to `LCS("CDE","GHI")` and another three calls to `LCS("BCDE","HI")`. Eventually we'll end up with a whopping 35 calls to `LCS("E","I")`. This explains why the algorithm is so slow—like Sisyphus, it keeps pushing the same rock up the same hill. Lather. Rinse. Repeat.

Be careful not to draw the wrong conclusion from this example. Sometimes people look at an example like this and say "See, recursion is slow!" However, the fault here lies in a stupid algorithm that needlessly repeats work, not in the fact that the stupid algorithm happens to be written recursively.

Somehow we need to stop the algorithm from repeating work that it has already done. That's where DP comes in. But first we'll take a short detour into a close cousin of DP called memoization.

Memoization

Memoization is the technique of remembering what inputs we've called a function on, and the answer for each of those inputs. Then, if we call the function a second (or 35th) time on the same inputs, we can simply lookup the answer instead of recomputing it. (And no, "memoization" is not a typo. It looks like it should have an "r" in the middle, but it doesn't.)

For example, here is the LCS algorithm re-written to use memoization. It depends on a "memo table" of answers indexed by pairs of strings:

```
function MLCS(S, T) is
if the pair <S,T> is in the memo table then
lookup and return the answer associated with the pair <S,T>
otherwise
```

```
answer = LCS(S,T)  // do the actual work for these inputs
save answer in memo table with the pair <S,T>
return answer

function LCS(S, T) is
if S is empty or T is empty then return empty string
if first char of S == first char of T then
return (first char of S) + MLCS(S - first char, T - first char)
otherwise // first chars are different
return longer of MLCS(S - first char, T) and MLCS(S, T - first char)
```

For clarity, I've isolated the memoization code into a separate function MLCS that calls the original LCS function. The only change to LCS is that it now calls MLCS instead of calling itself recursively. (Of course, LCS calls MLCS, which in turn calls LCS again, so there is still recursion going on here. But now it is mutual recursion between LCS and MLCS instead of between LCS and itself.)

In the code above, the memo table would probably be implemented as a hash table. This is a very common representation of memo tables, especially when the inputs are strings. In this particular case, however, we can do better, assuming we are willing to erase the memo table between distinct calls to the algorithm. Instead of passing around entire strings, we can pass around indices into those strings. Then the memo table can be a simple two-dimensional array indexed by integers. The catch is that we need arrays of different sizes whenever we call the main function with different initial strings. So we allocate a new array when we start the main function, and deallocate the array when we finish. We use an index of 0 to indicate that we are at the beginning of a string, and an index one past the last position in the string to indicate that we have reached the end of the string. The main function is now

```
function MAIN-LCS(S, T) is
allocate an array A[0..length of S, 0..length of T]
initialize all entries in A to null
answer = LCS(0,0)
deallocate A
return answer
```

with LCS and MLCS as local helper functions

```
function MLCS(i, j) is
if A[i,j] == null then
A[i,j] = LCS(i,j)
return A[i,j]

function LCS(i, j) is
if i == length of S or j == length of T then return empty string
if S[i] == T[j] then
return S[i] + MLCS(i+1, j)
otherwise
return longer of MLCS(i+1, j) and MLCS(i, j+1)
```

The memoization-based algorithm processes the memo table in a top-down, demand-driven fashion. Taking the final step to full-fledged DP involves processing the memo table bottom-up, instead.

Dynamic Programming...At Last

If you trace through the execution of the memoization-based algorithm, you'll see that it asks for answers to the biggest subproblems first. But if you look at the order in which answers are actually written into the memo table, you'll see that it fills in answers to the smallest subproblems first. This discrepancy arises from the LIFO nature of recursion—the first invocation of a recursive function is the last to return.

DP throws away the recursion and simply focuses on filling in the table, using a few loops to initialize the table from the smallest subproblems to the biggest subproblems. In fact, as a side note for those of you interested in etymology, the word "programming" in dynamic programming refers, not to computer programming, but to the family of mathematical techniques based on filling out and manipulating tables (linear programming is perhaps the most famous member of this family).

Converting the previous algorithm to use DP is straightforward, except for one point that I'll discuss in a moment.

```
function LCS(S, T) is
allocate an array A[0..length of S, 0..length of T]
for i = length of S downto 0 do
for j = length of T downto 0 do
if i == length of S or j == length of T then
A[i,j] = empty string
else if S[i] == T[j] then
A[i,j] = S[i] + A[i+1,j]
else
A[i,j] = longer of A[i+1,j] and A[i,j+1]
answer = A[0,0]
deallocate A
return answer
```

If you compare the body of the inner loop side-by-side with the body of the previous LCS function, you can see that the two algorithms really are doing the same thing.

DP	Memoization
`if i == length of S or j == length of T then`	`if i == length of S or j == length of T then`
`A[i,j] = empty string`	`return empty string`
`else if S[i] == T[j] then`	`if S[i] == T[j] then`
`A[i,j] = S[i] + A[i+1,j]`	`return S[i] + MLCS(i+1,j)`
`else`	`otherwise`
`A[i,j] = longer of A[i+1,j] and A[i,j+1]`	`return longer of MLCS(i+1,j) and MLCS(i,j+1)`

The main differences are that calls to MLCS have been replaced with lookups in the array, and the returns have been replaced with assignments into the array.

The most confusing part of the above DP algorithm is why the loops count backwards instead of forwards. This is actually an artifact of the way I wrote the recursive algorithm. In

the recursive algorithm, it was most natural to obtain subproblems by peeling off characters from the fronts of the strings. But that means that the smaller subproblems are at the backs of the strings, which in turn means that, if you want to process the table from smallest subproblems to biggest subproblems, you end up working backward.

With DP, however, it is probably more natural to work front to back. Fortunately, this is a very easy change to make.

```
function LCS(S, T) is
allocate an array A[0..length of S, 0..length of T]
for i = 0 upto length of S do
for j = 0 upto length of T do
if i == 0 or j == 0 then
A[i,j] = empty string
else if S[i-1] == T[j-1] then
A[i,j] = A[i-1,j] + S[i-1]
else
A[i,j] = longer of A[i-1,j] and A[i,j-1]
answer = A[length of S,length of T]
deallocate A
return answer
```

This answer is fine as is, but aficionados of DP often take the further step of initializing the base cases outside the main loop, as in the following version.

```
function LCS(S, T) is
allocate an array A[0..length of S, 0..length of T]

// initialize base cases
for i = 0 upto length of S do
A[i,0] = empty string
for j = 0 upto length of T do
A[0,j] = empty string

// main loop
for i = 1 upto length of S do
for j = 1 upto length of T do
if S[i-1] == T[j-1] then
A[i,j] = A[i-1,j] + S[i-1]
else
A[i,j] = longer of A[i-1,j] and A[i,j-1]

answer = A[length of S,length of T]
deallocate A
return answer
```

Another Example: Knapsack

There are many variations of the knapsack problem, but here's one: Given an array C[1..K] of distinct positive integers, count how many combinations of integers in C add up to exactly N.

We begin with a straightforward recursive solution, where combos(i,m) is the number of combinations of integers in C[1..i] that add up to m.

```
function combos(i,m) is
if m = 0 then return 1
if i = 0 then return 0
if C[i] > m then return combos(i-1,m)
else return combos(i-1,m) + combos(i-1,m-C[i])
```

In the first case, there is exactly one combination of integers that adds up to 0, namely the empty combination. In the second case, we have no numbers to add so there is no way to get a non-zero sum. In the third case, C[i] is too big so the only way to add up to m is to use the integers in C[1..i]. In the last case, we count the number of combinations that do not use C[i] plus the number of combinations that do use C[i].

Notice that i is always between 0 and K, inclusive, and m is always between 0 and N, inclusive. Therefore, in the DP solution, we use an array A[0..K, 0..N]. We initialize the array from smaller values of i and m to bigger values.

```
function combos(C) is
allocate an array A[0..K, 0..N]

// initialize base cases
A[0,0] = 1
for m = 1 upto N do
A[0,m] = 0

// main loop
for i = 1 upto K do
for m = 1 upto M do
if C[i] > m then
A[i,m] = A[i-1,m]
else
A[i,m] = A[i-1,m] + A[i-1,m-C[i]]

answer = A[K,N]
deallocate A
return answer
```

This algorithm obviously takes O(K·N) time and space. With a small amount of cleverness, we can reduce the space requirement to O(N). The key is to realize that each row of the table depends only on the previous row. Therefore, we only need to keep one row in memory at a time.

With that hint, most people will go out and write the following code:

```
function combos(C) is // Warning: Buggy!
allocate an array A[0..N]

// initialize base cases
A[0] = 1
for m = 1 upto N do
A[m] = 0

// main loop
for i = 1 upto K do
for m = 1 upto N do
// if C[i] > m then A[m] is unchanged
if C[i] <= m then
A[m] = A[m] + A[m-C[i]]

answer = A[N]
deallocate A
return answer
```

Unfortunately, there's a serious bug in this code that illustrates how important it is to be careful about the order in which you fill in the table. At each iteration of the inner loop, A[1..m-1] has already been updated to hold the number of valid combinations of integers from C[1..i], whereas A[m..N] still holds the number of valid combinations of integers from C[1..i-1]. Now, look at the innermost assignment

```
A[m] = A[m] + A[m-C[i]]
```

By the time we do this assignment, we've already updated A[m-C[i]], whereas what we want here is the previous value of A[m-C[i]]. The fix is simply to run the inner loop backwards from N to 1 instead of forwards from 1 to N. That way, when we process A[m], we have not yet processed A[m-C[i]]. The corrected code is

```
function combos(C) is
allocate an array A[0..N]

// initialize base cases
A[0] = 1
for m = 1 upto N do
A[0,m] = 0

// main loop
for i = 1 upto K do
for m = N downto 1 do
```

```
// if C[i] > m then A[m] is unchanged
if C[i] <= m then
A[m] = A[m] + A[m-C[i]]

answer = A[N]
deallocate A
return answer
```

Summary

Coming up with a DP algorithm involves three main steps. Once you get good at it, you won't have to think about the steps separately, but in the beginning, it's best to go through the steps one at a time.

- **Come up with the recursive relationship.** Many beginning programmers get hung up here. If you have troubles in this step, set aside DP for awhile and go practice recursion.
- **Look for overlapping subproblems.** If there are no overlapping subproblems, then you don't need DP. If there are overlapping subproblems, see if you can characterize the subproblems with a few numbers chosen from a small range so that you can allocate an array of the right size. If you can't figure out a range of numbers that works, you can use a less structured table like a hashtable instead of an array. But if you go that route, be careful because there may end up being many more subproblems than you expected.
- **Figure out the right order to fill in the table.** Make sure that each element of the table is initialized before it is needed in the calculation of another element. Keep an open mind. Sometimes your loops will go forwards, sometimes backwards, and sometimes you'll have a mix of both. If you get stuck on this step, but you've completed the first two, then bail on DP and just go with memoization.

As with most programming skills, the best way to learn DP is to practice, practice, practice. Fortunately, the practice rooms have a wide variety of DP problems to choose from. For easy practice problems, try AvoidRoads (2003 TCO Semifinal Room 4) or ZigZag (2003 TCCC Semifinal Room 3). When you're ready for more challenging problems, try Jewelry (2003 TCO Online Round 4) or StripePainter (SRM 150 Div 1).

Good luck, and happy DP-ing!

Would you like to write a feature?

PRIVACY POLICY

TERMS

topcoder is also on